Experiences in Parallel Debugging

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

November 10, 1993

1 Introduction

This document describes a prototype thread and lock debugging package that has been used for debugging parallelized *Tera* file system code. One of the initiatives behind this prototyping is to provide an environment to experiment with various facilities that could be useful for parallel debugging and as a result facilitate more thorough parallel testings prior to first shipment of the *Tera* operating system code.

The following is a list of features provided by this debugging package:

1. provide a global picture of all threads that are currently holding or waiting for a lock
2. provide a global picture of all current locks that a thread owns
3. report cycles if a lock violates an existing locking hierarchy
4. generate a linear graph describing the parent child relationships among locks

In the next 4 sections we describe each of the above features and finally discuss the use of these features for debugging parallelized user programs.

2 Locks

An object `lock` is an instance of a `Tera OS` synchronization primitive. Associated with each `lock` is a linked list of `threads` that are currently waiting or holding the `lock`. The full empty bit of a `sync int` location is the underlying mechanism for implementing locking and unlocking. When a `thread` attempts to acquire a `lock` it first inserts itself into the list of threads waiting on the `lock`. When this thread successfully acquires the `lock`, ownership of the `lock` is then set to itself.

As a `thread` completes its critical section and is ready to give up a previously acquired `lock`, it first removes itself from the list of waiting `threads` and then releases the `lock`.

During debugging this information allows us to observe all threads that are currently blocked or holding a given `lock`.

A sample printout from invoking `lockA->printLock()` is as follows.

```
 SpinLock:
     name=LockA, initialized=TRUE
     List of threads waiting or holding LockA:
      1  3  6
 owner = 1
```

In this example a `lock` named `LockA` is currently owned by thread id 1. The type of `lock` is a mutual exclusive `SpinLock`. As it is shown, there are two other threads 3 and 6 which are currently blocked waiting to acquire `LockA`.

3 Threads

An object `thread` is an instance of a parallel thread of control. `Threads` use `locks` to enforce serialization of critical sections of code. Associated with each `thread` is a list of `locks` that are currently owned or waited on by this `thread`. When a thread attempts to acquire a `lock` it inserts this `lock` into its `lock list`. After a `thread` acquires the `lock` it claims ownership of the respective `lock`.

At any point in time during interactive debugging this information allows us to determine all `lock` resources that a `thread` owns or is blocked waiting for its availability.

A sample printout after calling `thread1->printThread()` is as follows.

```
 thread id 1: foo2

 SpinLock:
     name=LockA, initialized=TRUE
     List of threads waiting or holding LockA:
      1   3   6
 owner = 1

 SpinSuspendLock:
     name=suspendingLock, initialized=TRUE
     List of threads waiting or holding suspendingLock:
      1
 owner = 1
```

This example shows that the current `thread` has an id of 1. The name of its invoking function is `foo2`. Currently `thread 1` owns 2 `locks`. It has acquired `suspendingLock` first and then `LockA` later. Note that the `printThread()` method invokes the `printLock()` method.

Therefore, during an interactive debugging session, one can invoke `printAllThread()` to print out a global picture of all active `threads` in the system. In addition to this global picture the stack traces of all **threads** provide a complete snapshot of the entire executing parallel program. With this information the programmer could identify any potential deadlock in the system.

The following example shows the printout of `printAllThread()` and the trace stack of all **threads**. In this example `thread 1` held `suspendingLock` and `lockA`, while `thread 2` is blocked waiting on `lockA`.

```
===========Print All Threads=====================


thread id 1: foo2

SpinLock:
    name=LockA, initialized=TRUE
    List of threads waiting or holding LockA:
     1   2
owner = 1

SpinSuspendLock:
    name=suspendingLock, initialized=TRUE
    List of threads waiting or holding suspendingLock:
     1
owner = 1

---------------------------


thread id 2: foo4

SpinLock:
    name=LockA, initialized=TRUE
    List of threads waiting or holding LockA:
     1   2
owner = 1
```

```
===============gdb output==============================

(gdb) tdump
Frames for thread_number=0, thread->context()=0x000e2620 follow
(p=0x000ea3e0, next=0x000f2f20, prev=0x000fad20)
{0, 0}: (sp=0x000ea040) _reschedule__6CpuMuxP6Thread

Frames for thread_number=1, thread->context()=0x000f2f60 follow
(p=0x000fad20, next=0x000ea3e0, prev=0x000f2f20)
{1, 0}: (sp=0x000faaf8) _foo2
{1, 1}: (sp=0x000fab78) ParallelThread::_main()
{1,2}: (sp=0x000fabe8) _startOff_6ThreadPT0
{1,3}: (sp=0x000fac58) Thread::_main()

Frames for thread_number=2, thread->context()=0x000eb160 follow
(p=0x000f2f20, next=0x000fad20, prev=0x000ea3e0)
{2, 0}: (sp=0x000f2928) _reschedule__6CpuMuxP6Thread
{2, 1}: (sp=0x000f2998) CpuMux::_reschedule(Thread *)
{2, 2}: (sp=0x000f2a08) _yieldThread
{2, 3}: (sp=0x000f2a80) _sync_int_read
{2, 4}: (sp=0x000f2af0) _updateHier_8HierLockPT0
{2, 5}: (sp=0x000f2b70) _LockDebug_k::_insert(ParallelThread *, int)
{2, 6}: (sp=0x000f2c00) SpinLock_k::_lock(int)
{2, 7}: (sp=0x000f2c88) _Spin_lock
{2, 8}: (sp=0x000f2cf8) _foo4
{2, 9}: (sp=0x000f2d78) ParallelThread::_main()
{2,10}: (sp=0x000f2de8) _startOff_6ThreadPT0
{2,11}: (sp=0x000f2e58) Thread::_main()
```

4 Cycles and Deadlocks

During runtime, associated with each `lock` is a history of the hierarchy in which `locks` have been taken. Every time a `lock` is acquired by a `thread`, its most recently acquired `lock` if any will be examined and a relationship is updated between the most recently acquired `lock` and the newly to be acquired `lock`. For example, a `thread` is in the process of acquiring a `lock` named `LockB`. The last `lock` that this `thread` owns is a `lock` named `LockA`. So, we update the locking hierarchy of `LockA` and `LockB` so that one of the parents of `LockB` is `LockA` and one of the children of `LockA` is `LockB`. If during some prior locking either by this or some other thread this newly derived relationship contradicts an existing relationship a cycle is reported, routine `printHier()` is called, and the program terminates. This allows corrective action to be taken by the programmer before a deadlock actually takes place.

An example printout below shows the current parent child relationships among a program's `locks` after explicitly calling `printHier()`.

```
0) LockA - 2 P: 2  3 ,    1 C:  1

1) LockB - 1 P: 0 ,    0 C:

2) suspendingLock - 0 P: ,   1 C:  0

3) readerWriterLock - 0 P: ,   1 C:  0
```

In this example, the `LockA` has two parents; `suspendingLock` and `readerWriterLock`. Therefore, `suspendingLock` and `readerWriterLock` show that they both have `LockA` as their child. This set of relationships enforces a linear ordering of `locks`. If a cycle is detected it notifies you prior to occurrence of a deadlock. Note that when `tryLock()` is used when a given ordering cannot be observed and succeed in obtaining a `lock`, cycle detection will not be activated.

5 Hierarchy

Using the parent child relationships generated above, a linear graph of all `locks` is produced. The heuristics for generating a linear graph consist of setting up in linear order a doubly linked hierarchical list that maintains the parent child relationships among all `locks`.

Its algorithm is as follows: In a loop examine each `lock` in the entire system.

1. Examine each parent of the `lock`
2. Is parent already on the hierarchy list?
3. If so, skip this step. Otherwise, construct a new hierarchy node and insert after the parent's parents
4. Examine the `lock` itself
5. Is `lock` already on the hierarchy list?
6. If so, skip this step. Otherwise, construct a new hierarchy node and insert at the head of the list
7. Is the `lock` node placed after all its parents?
8. If so, skip this step. Otherwise, insert it right after its last parent
9. Examine each child of the `lock`
10. Is child already on the hierarchy list?
11. If so, skip this step. Otherwise, insert it right after its last parent
12. Loop until all `locks` are examined

A linear graph generated from the parent child relationships of the previous example is as follows.

```
Linear Graph of Locks:
 3,  2,  0,  1
```

The above example is interpreted as: `readerWriterLock` (3) must be acquired before `suspendingLock` (2) which must be acquired before `LockA` (0), and so on. Note that this linear graph is not unique. In fact, a linear ordering of (2, 3, 0, 1) is also valid.

A linear ordering of `locks` guarantees cycle free locking. It provides a *map* when changing or adding into existing parallelized code.

Figure 1 shows the relationships between all the different data structures as described in this document.

6 Sync, Future, and Discussions

This package has already benefited the debugging of parallelized file system code. Interesting enough the Solaris 2 parallel `dbx` promises some of the similar features for `lock` and `thread` in their future GA release of their parallel `dbx`.

This package has been made thread-safe and runs under Awesim and zebra. Initially it is designed, implemented, and tested for the file system unit test environment. Future objectives are to incorporate the package into the Tera OS integrated build environment.

I have used this package so far to identify race and deadlock conditions within the Unix file system code. Several test cases were generated for testing parallel activities within the file system.

Even though the above discussions were all centered around `locks` and `threads` for the `Tera OS`, similar benefits can be realized in user applications using `sync ints` and `futures`. In fact, the `locks` and `threads` implemented under `zebra` for the file system unit test environment have been based on `sync ints`, `futures`, and the `Tera` user run time environment.

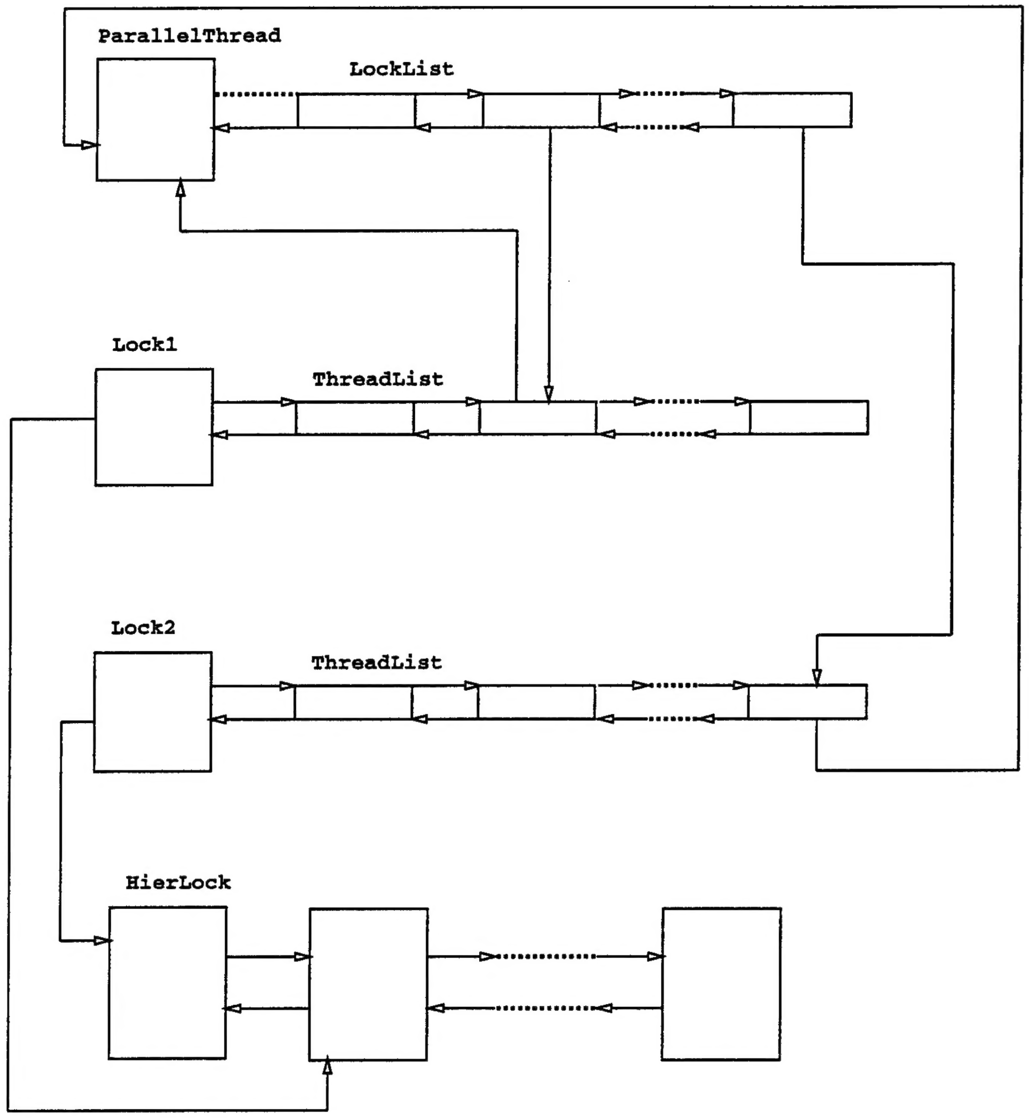

FIGURE 1: Threads and Locks Relationship